

## Branch-and-bound for TSP

In this tutorial, we consider the Traveling Salesman Problem. We formulate the problem in terms of weighted graphs rather than in terms of matrices as in the lecture. A *weighted graph*  $G$  is a graph where each edge carries a weight in  $\mathbb{R}$ . A *Hamiltonian circuit* of  $G$  is a cycle of edges of  $G$  that visits each vertex exactly once. The cost of such a circuit is the sum of weights of its edges. The TSP problem is the optimization problem that consists in finding a Hamiltonian circuit of minimal cost (if no Hamiltonian circuit exists, the answer is  $+\infty$ ). For example Figure 1 shows a weighted graph on the left side, and the optimal Hamiltonian circuit (which has cost 8) on the right side.<sup>1</sup>

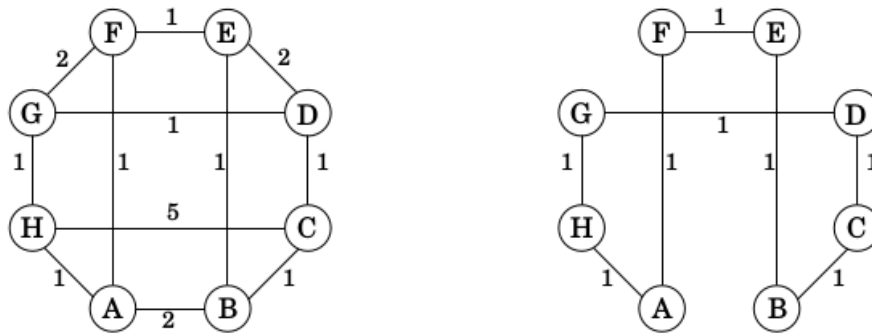


Figure 1: A weighted graph (left) and the optimal Hamiltonian circuit of cost 8 (right)

Download the files `WG.py`, `test.py` and `cities.txt`. The latter contains a matrix of distances between European cities to be used for testing. The former contains two classes:

- a UnionFind class `UF` that is used by the minimal spanning tree method
- the `WG` class representing a weighted graph

All the code to be completed is in the class `WG` (file `WG.py`). The class has a constructor that receives a list of edges with their weights in the form

```
[['a','b',1],['c','b',2],['a','c',4]].
```

For instance, in this example, `a` and `c` are connected by an edge of weight 4. The list is stored in the attribute `self.edges`, sorted by edge-weights, to facilitate the computation of minimum spanning trees. The constructor also builds an attribute `adj` which is a dictionary associating to vertices their neighbors and the weights of edges between them. In particular:

- testing if `x` is a vertex of `self` is done by `if x in self.adj`
- if `x`, `y` are two vertices, testing if `x` and `y` are adjacent is done by `if y in self.adj[x]`
- if `x`, `y` are two *adjacent* vertices, `self.adj[x][y]` gives the weight of the edge connecting them

### 1 A brute-force recursive algorithm

**Question 1.** The method `min_cycle_aux(self,w,L,S)` has the following parameters: `L` is a list of vertices (of length at least 1) that forms a path in the graph, `w` is the total weight of edges on this

<sup>1</sup>The two figures are taken from Section 9.1.2 of the book “Algorithms” by Dasgupta, Papadimitriou, Vazirani, which we closely follow.

path, and  $S$  is the set (Python `set` structure) of vertices that are not in  $L$ . The method has to return a pair  $(W, \text{Cyc})$  where  $\text{Cyc}$  is a Hamiltonian circuit of smallest cost among the Hamiltonian circuits that start with the path  $L$ ; and  $W$  is the cost of  $\text{Cyc}$ . For instance, in the example of Figure 1, if we take  $L=[A, F, G]$ , then there are two Hamiltonian circuits that extend  $L$ : the circuit  $[A, F, G, D, E, B, C, H, A]$  of cost 14, and the circuit  $[A, F, G, H, C, D, E, B, A]$  of cost 15. Hence the method has to return the pair  $(14, [A, F, G, D, E, B, C, H, A])$ . If there is no Hamiltonian circuit starting with  $L$ , return  $(\text{math.inf}, [])$ . Complete the method `min_cycle_aux(self, w, L, S)` (which has to proceed recursively). To test your method execute `test1()`.

**Question 2.** Then complete the method `min_cycle(self)`, which has to return a pair  $(W, \text{Cyc})$  where  $\text{Cyc}$  is a Hamiltonian circuit of smallest cost, and  $W$  is its cost (it can help to write an auxiliary method that returns the set of vertices of the graph). To test your method execute `test2()`.

## 2 Improvement using branch-and-bound

Figure 2 shows the exploration tree for the graph of Figure 1 starting from vertex  $A$ . Alongside each node in that exploration tree—corresponding to a path  $L$  starting from  $A$ —the framed number indicates the lower bound on the cost of a Hamiltonian circuit starting with  $L$ , as we define it in the following. The sub-tree rooted in the node is not explored if that lower bound is at least as large as the minimal cost of a Hamiltonian circuit seen so far.

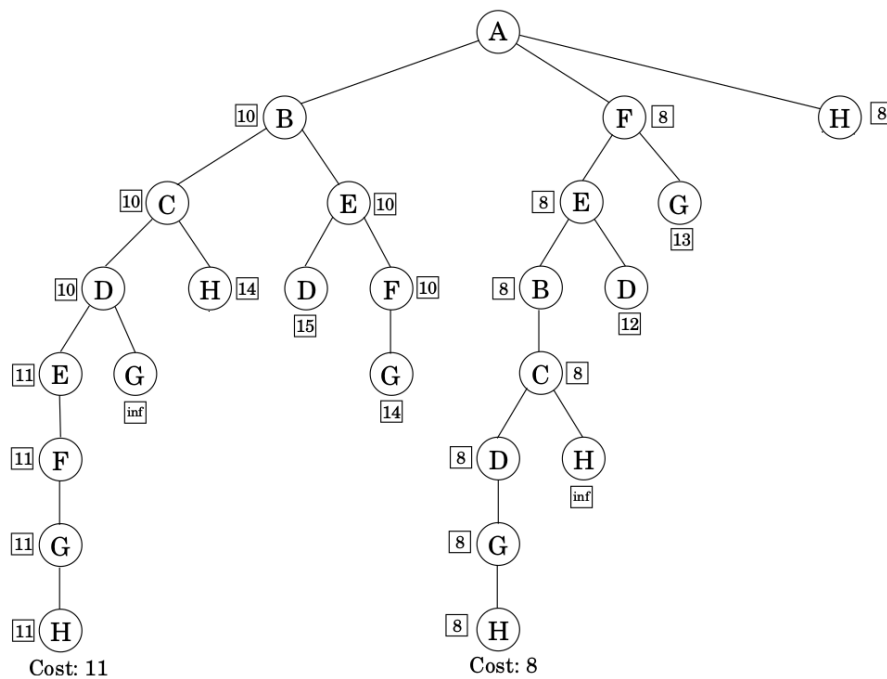


Figure 2: The exploration tree for the graph of Figure 1 starting from vertex  $A$

**Question 3** (Exercise). With the notation of Question 1, and assuming  $S$  is not empty, let  $T_S$  be a spanning tree of minimal cost among the trees whose spanned vertices are those of  $S$ , and let  $w_S$  be the cost of  $T_S$  ( $w_S = +\infty$  if there is no such tree, i.e., if the graph restricted to  $S$  is not connected). Let  $w_{\text{start}}$  be the minimal weight over edges from  $L[0]$  to vertices in  $S$  ( $w_{\text{start}} = +\infty$  if there is no such edge), and let  $w_{\text{end}}$  be the minimal weight over edges from  $L[-1]$  to vertices in  $S$  ( $w_{\text{end}} = +\infty$  if there is no such edge). Show that the weight  $W$  of the output circuit  $\text{Cyc}$  satisfies

$$W \geq w + w_{\text{start}} + w_{\text{end}} + w_S.$$

(Hint: note that a path connecting all vertices of  $S$  is a special kind of spanning tree on  $S$ .)

In other words, if we let  $low(L)$  be the quantity  $w + w_{start} + w_{end} + w_S$ , then  $low(L)$  provides a *lower bound* on the value of any circuit that extends  $L$ , and moreover this lower bound can be computed fast (we use here Kruskal's algorithm to compute minimal spanning trees). The recursive method in Question 1 can be represented as the exploration of a tree: starting the exploration from a vertex (vertex  $A$  for instance), each call to `min_cycle_aux` corresponds to a node of the tree, and the path  $L$  from the root to the node corresponds to the list  $L$  that is the parameter of `min_cycle_aux` for that call. A crucial observation is that if, during the exploration, we are at a node  $L$ , such that  $low(L)$  is at least as large as the mincost over Hamiltonian circuits seen so far, then we can abort the exploration at that node (i.e. avoid exploring the sub-tree rooted at  $L$ ). This is illustrated in Figure 2.

**Question 4.** Complete the method `lower_bound(self,w,L,S)`, which computes  $low(L)$  (with  $w$  the cost of  $L$  and  $S$  the set of vertices that are not in  $L$ ; we assume  $S$  is not empty). You will have to use the method `weight_min_tree(self,S)` in the class `WG` (this method returns the quantity  $w_S$ ). To test your method execute `test4()`.

In addition to the lower bounds, shown in Figure 2 as framed numbers, two of the nodes in the figure are annotated with cost values. This value, which we call *bestsofar*, is initialised to  $+\infty$ , and we only update it once we find a better solution. For example, in the figure, *bestsofar* remains  $+\infty$  until we reach the node representing  $L = [A, B, C, D, E, F, G, H]$ , where it becomes 11. It then remains 11 until we reach the node for  $L = [A, F, E, B, C, D, G, H]$ , where it becomes 8 (the tree is explored according to a depth-first search from left to right). No better Hamiltonian circuits are found during the exploration. Hence, we conclude, that  $[A, F, E, B, C, D, G, H, A]$  is an optimal Hamiltonian circuit.

**Question 5.** Complete the method `min_cycle_aux_using_bound(self,bestsofar,w,L,S)`. It has to return `(math.inf, [])` if  $low(L) \geq bestsofar$ . Otherwise, it has to return the same output as the method `min_cycle_aux(self,w,L,S)` while implementing the branch-and-bound technique illustrated in Figure 2. We assume that the method is called with the value of `bestsofar` giving the mincost over Hamiltonian circuits seen so far (i.e. over the nodes visited before  $L$ ), accordingly you have to correctly update the value of `bestsofar` for the recursive calls to `min_cycle_aux_using_bound` in your code. Then complete the method `min_cycle_using_bound(self)`, which has to return the same output as `min_cycle(self)`.

To test your method, you can use two methods: `test_random_graph(n)`, which tests both methods `min_cycle` and `min_cycle_using_bound` (and compares their running times), for a complete graph on  $n$  vertices where the edge weights are (uniformly) random in  $[0, 1]$ . The other method is `test_trip()`, which runs `min_cycle_using_bound` to determine the shortest circuit to visit a selected set of cities in Europe (among a fixed list of cities).

**Remark.** To evaluate the efficiency of the branch-and-bound technique, you can also introduce a global variable `nr_calls` and use it to count the total number of (recursive) calls to `min_cycle_aux` during the execution of `min_cycle`. Doing the same for the methods using bounds, you can then compare the number of calls in `min_cycle` and in `min_cycle_using_bound` for random graphs with  $n$  vertices.

**Remark.** The efficiency of the branch-and-bound technique crucially depends on the quality of the lower bounds, and there is a delicate trade-off (finding a lower bound that is good and not too time-consuming to compute). In our case, we can generalize the lower bounds based on the following simple observation due to Held and Karp: if we fix any function  $\pi: V \rightarrow \mathbb{R}$  (with  $V$  the vertex set) and update the weight of each edge  $e = (u, v)$  to be  $w_\pi(e) = w(e) + \pi(u) + \pi(v)$ , then the optimal values are clearly related by  $TSP = TSP_\pi - 2 \sum_{v \in V} \pi(v)$ . If we are at a certain node of the exploration tree, we can use this observation and look for functions  $\pi$  that will give us better and better lower bounds (note that the optimal spanning tree on vertices of  $S$  depends on the function  $\pi$  that is chosen). This can be efficiently done by an iteration updating  $\pi$  (improving the lower bound) progressively, based on a gradient computation.

Several other clever techniques have been developed to gain on time complexity (looking for local optima under certain local operations to update the circuit, Linear Programming relaxation combined with branch-and-bound techniques etc.), see for instance Section 9.10.2 of the book “The nature of computation” by Moore & Mertens for a clear presentation of such techniques. This allows to solve the TSP problem on some very large graphs, e.g. computing the shortest tour through all 24 978 cities of Sweden.